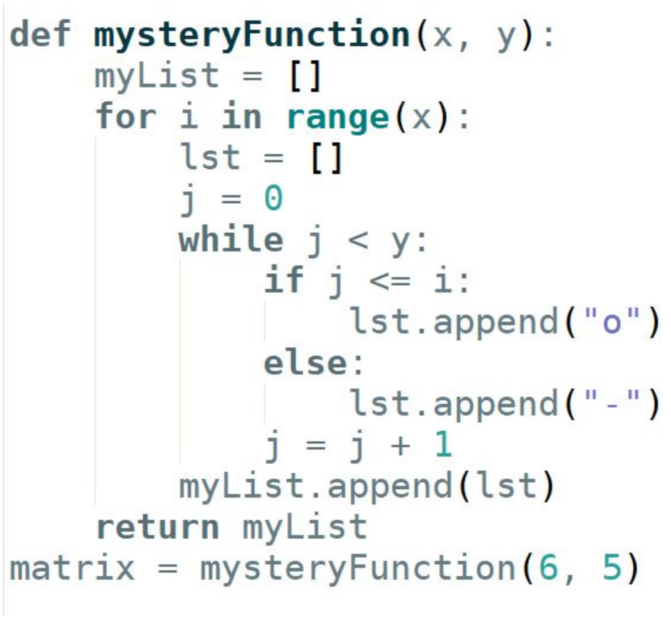
15-110 Hw3 - Written Portion

Name:

AndrewID:

#1 - 2D Lists - 5pts

Fill in the following table with the values in the 2D list returned by `mysteryFunction`. Write an X in the squares that are outside the bounds of the list.



```
def mysteryFunction(x, y):
    myList = []
    for i in range(x):
        lst = []
        j = 0
        while j < y:
            if j <= i:
                lst.append("o")
            else:
                lst.append("-")
            j = j + 1
        myList.append(lst)
    return myList
matrix = mysteryFunction(6, 5)
```


#2 - Recursion Tracing - 5pts

Trace the following code, then fill out the table below to indicate all the **recursive function calls** that are made, and which **value** is returned by each function call. You may not need all of the rows.

```
def gcd(x, y):
    if y == 0:
        return x
    return gcd(y, x % y)

print(gcd(20, 12))
```

Note: in the second column, make sure to indicate the actual returned **value**, not a function call or an expression!

Function Call	Returned Value
`gcd(20, 12)`	

#3 - Tracing Towers of Hanoi - 6pts

Recall the algorithm we discussed in class to solve the Towers of Hanoi problem. Use that algorithm to fill out all the steps needed to move three discs from Peg A to Peg C in the table below. You might not need to use all the rows.

The three discs are called 1, 2, 3 (where 1 is the smallest and the disc on top). So the algorithm starts with the discs 1, 2, 3 on Peg A, and should end with 1, 2, 3 on Peg C. We've done the first step for you.

	Peg A	**Peg B**	**Peg C**
Start	1, 2, 3		
Step 1	2, 3		1
Step 2			
Step 3			
Step 4			
Step 5			
Step 6			
Step 7			
Step 8			
Step 9			
Step 10			
Step 11			

How many steps would it take to move 4 discs instead of 3?

#4 - Linear Search Debugging - 8pts

The following three functions all attempt to implement the algorithm linear search, but with a twist: instead of identifying whether or not the target occurs in the list, each function returns the first index where the item occurs, or -1 if it never shows up. However, only one of the three is correct. Identify which of the three functions is correct, then explain what is wrong with the other two and how they can be fixed.

```
def linearSearchA(lst, target):
    for i in range(len(lst)+1):
        if lst[i] == target:
            return i
    return -1

def linearSearchB(lst, target):
    i = 0
    while i < len(lst):
        if lst[i] == target:
            return i
        i = i + 1
    return -1

def linearSearchC(lst, target):
    if len(lst) == 0:
        return -1
    elif lst[0] == target:
        return 0
    else:
        return 1 + linearSearchC(lst[1:], target)
```

Which implementation is correct?

- ☐ linearSearchA
- ☐ linearSearchB
- ☐ linearSearchC

Why are the other two incorrect, and how can they be fixed?

#5 - Binary Search - 6pts

In the following table, write out the recursive calls that our implementation of `binarySearch` would make while searching the given list for the given item. Make sure to write out the **function call**, not the result. You might not need to use all the rows.

Q1: Search for 5

Original Call	`binarySearch([3, 5, 6, 7, 9, 11, 11, 15, 15, 19], 5)`
Recursive Call 1	
Recursive Call 2	
Recursive Call 3	
Recursive Call 4	
Recursive Call 5	

Q2: Search for 14

Original Call	`binarySearch([3, 5, 6, 7, 9, 11, 11, 15, 15, 19], 14)`
Recursive Call 1	
Recursive Call 2	
Recursive Call 3	
Recursive Call 4	
Recursive Call 5	

#6 - Best Case and Worst Case - 10pts

For each of the following functions, describe an input that would result in **best-case efficiency,** then describe an input that would result in **worst-case efficiency**. This generic input must work at **any possible size**; don't answer 1 for `isPrime`, for example.

```python
def getEmail(words):
    # words is a list
    for i in range(len(words)):
        if "@" in words[i]:
            return words[i]
    return "No email found"
```

```python
def isPrime(num):
    for factor in range(2, num):
        if num % factor == 0:
            return False
    return True
```

What is a **best case input** for `getEmail`?

What is a **worst case input** for `getEmail`?

What is a **best case input** for `isPrime`?

What is a **worst case input** for `isPrime`?

#7 - Calculating Big-O Families - 15pts

For each of the following functions, check the **Big-O function family** that function belongs to. You should determine the function family by considering how the number of steps the algorithm takes grows as the size of the input grows.

```
def countEven(L): # n = len(L)
  result = 0
  for i in range(len(L)):
    if L[i] % 2 == 0:
      result = result + 1
  return result
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

```
# n = len(L)
def sumFirstTwo(L):
  if len(L) < 2:
    return 0
  return L[0] + L[1]
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

```
# n = len(L1) = len(L2)
def linearSearchAll(L1, L2):
  count = 0
  for item in L1:
    # Hint: what's the complexity of
    # linear search?
    if linearSearch(L2, item) == True:
      count = count + 1
  return count
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

```
# n = len(L1) = len(L2)
def binarySearchAll(L1, L2):
  count = 0
  for item in L1:
    # Hint: what's the complexity of
    # binary search?
    if binarySearch(L2, item) == True:
      count = count + 1
  return count
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

```
# n = len(L); original call has i = 0
def recursiveSum(L, i):
  if i == len(L):
    return 0
  else:
    return L[0] + recursiveSum(L, i+1)
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

#8 - Tracing Sorting Algorithms - 20pts

For the two tables below, each row represents a 'pass' - a single iteration of the outer loop in the function. Fill in the number of comparisons and swaps that happen in each pass, and the state of the list at the **end** of that pass, for the specified sort function as implemented in class.

Selection Sort			
Pass #	**Comparisons**	**Swaps**	**List State**
Start	-	-	**[** 3, 5, 1, 2, 4 **]**
1			
2			
3			
4			

Insertion Sort			
Pass #	**Comparisons**	**Swaps**	**List State**
Start	-	-	**[** 3, 5, 1, 2, 4 **]**
1			
2			
3			
4			

Merge Sort:
How many times is the function `mergeSort` called when we run `mergeSort([ 3, 5, 1, 2, 4 ])`?

15-110 Hw3 - Programming Portion

Each of these problems should be solved in the starter file available on the course website. Submit your code to the Gradescope assignment Hw3 - Programming for autograding.

All programming problems may also be checked by running the starter file, which calls the function `testAll()` to run test cases on all programs.

#1 - `addToAll(lst, x)` - 5pts

Write the function `addToAll(lst, x)` which takes a list of numbers and a number and **destructively** modifies the list so that every element has x added to it. For example, if `lst = [1, 2, 3]`, calling the function `addToAll(lst, 2)` will change `lst` to hold `[3, 4, 5]`.

#2 - `letterFrequency(s)` - 5pts

Write a function `letterFrequency(s)` which takes a string and returns a list of 26 elements, where each element is the number of times that the corresponding letter of the alphabet occurs in the string. The 0th index corresponds to "a", the 1st corresponds to "b", etc., until the 25th element corresponds to "z".

For example, `letterFrequency("Hello World")` should return: `[0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0]`.

Note: you can ignore any non-letter characters that occur in the string, but you should make sure both upper- and lower-case letters are counted as the same.

Hint: the easiest way to get an index based on a letter is to use the `ord(c)` method, which takes a one-character string as input and returns the ASCII value of that character. Offset this number by `ord("a")` or `ord("A")` to get the index you need.

#3 - `onlyPositive(lst)` - 5pts

Write a function `onlyPositive(lst)` that takes as input a 2D list and returns a new 1D list that contains only the positive elements of the original list, in the order they originally occurred. You may assume the list only has numbers in it.

Example: `onlyPositive([[1, 2, 3], [4, 5, 6]])` returns `[1, 2, 3, 4, 5, 6]`, `onlyPositive([[0, 1, 2], [-2, -1, 0], [10, 9, -9]]` returns `[1, 2, 10, 9]`, and `onlyPositive([[-4, -3], [-2, -1]])` returns `[ ]`.

#4 - `recursiveCount(lst, item)` - 5pts

Write a function `recursiveCount(lst, item)` that takes a list and a value as input and returns a count of the number of times that item occurs in the list. This function must use **recursion** in a meaningful way; a solution that uses a loop or built-in count functions will receive no points.

For example, `recursiveCount([2, 4, 6, 8, 10], 6)` returns `1`, `recursiveCount([4, 4, 8, 4], 4)` returns `3`, and `recursiveCount([1, 2, 3, 4], 5)` returns `0`.

#5 - `recursiveMax(lst)` - 5pts

Write a function `recursiveMax(lst)` that takes a list as input and returns the maximum value in the list. You may assume the list contains at least one element. This function must use **recursion** in a meaningful way; a solution that uses a loop or built-in max functions will receive no points.

For example, `recursiveMax([1, 2, 3])` returns `3`, and `recursiveMax([2, 4, 6, 9, 10, 2, 6])` returns `10`.

Hint: consider what properties the recursive result has if the function works as expected.